Inferring Algorithmic Patterns with Stack-Augmented Recurrent Nets

Armand Joulin AJOULIN@FB.COM
Tomas Mikolov TMIKOLOV@FB.COM
Facebook AI Research, 770 Broadway, New York, USA.

Abstract

While machine learning is currently very successful in several application domains, we are still very far from achieving a real artificial intelligence. In this paper, we study basic sequence prediction problems that are beyond the scope of what is learnable with popular methods such as recurrent networks. We show that simple algorithms can be learned from sequential data with a recurrent network associated with trainable stacks. We focus our study on algorithmically generated sequences such as $a^n b^n$, that can only be learned by models which have the capacity to count. Once trained, we show that our method is able generalize to sequences up to an arbitrary size. We discuss the limitations of standard machine learning approaches to learn algorithimic regularities of this type. We propose directions to overcome these shortcomings, such as using search based optimization.

1. Introduction

Machine learning aims to find regularities in data to perform various tasks. History shows that there are two major sources of breakthroughs: scaling up the existing approaches to larger datasets, and development of novel, more accurate approaches (LeCun et al., 1998; Elman, 1990; Breiman, 2001). In the recent years, a lot of progress has been made in scaling up algorithms, by either using alternative hardware such as GPUs (Ciresan et al., 2011) or by taking advantage of large clusters (Recht et al., 2011). While improving the speed of current methods is crucial to work on the very large datasets currently available (Bottou, 2010), it is also vital to develop novel approaches able to tackle new problems.

Recently, deep neural networks have become very successful at various tasks, leading to a shift in the computer vision (Krizhevsky et al., 2012) and speech recognition communities (Dahl et al., 2012). This breakthrough is commonly attributed to two aspects of deep networks: their similarity to the hierarchical, recurrent structure of the neocortex and the theoretical justification that certain patterns are more efficiently represented by functions employing multiple non-linearities instead of a single one (Minsky & Seymour, 1969; Bengio & LeCun, 2007).

It is important to question which patterns are difficult to represent and learn with the current state of the art methods. This would potentially give us hints to design new approaches which will hopefully advance machine learning research further. In the past, this approach was common and lead to crucial results: the well-known XOR problem is an example of trivial classification problem that cannot be solved using linear classifiers, but can be solved with a non-linear one. This lead to the use of non-linear hidden layers (Rumelhart et al., 1985) and kernels methods (Bishop, 2006). Another example is the parity problem introduced by Minsky & Seymour (1969): it demonstrates that while a single non-linearity is sufficient to represent any function, it is not guaranteed to represent it efficiently, and in some cases can even require exponentially many more parameters (and thus, also training data). This lead to the use of architectures that have several layers of non-linearities, currently known as deep models.

Following this line of work, we study basic patterns which are difficult to represent and learn for deep models. In particular, we study simple sequences of symbols generated from simple algorithms. Interestingly, we find that such patterns are difficult to learn even for some advanced deep learning models, such as recurrent networks. We attempt to increase the learning capabilities of recurrent nets by allowing them to learn structured memory, similar to pushdown stack that is widely used to parse context free languages.

An example of very simple problem of this type is a sequence $a^n b^n$, i.e., a sequence where the regularity is in the equal number of symbols a and b. A model that solves this task must be able to generalize for any reasonable n, i.e., after training on sequences up to some fixed length, our model should be able to recognize longer sequences generated from the same algorithm.

While these patterns seem relatively basic, solving them

Sequence generator	Example
$\{a^n b^n \mid n > 0\}$	aab**ba**aab**bba**b**a**aaaab**bbbb**
$\{a^n b^n c^n \mid n > 0\}$	aaab**bbccca**b**ca**aaaab**bbbbccccc**
$\{a^n b^n c^n d^n \mid n > 0\}$	aab**bccdda**aab**bbcccddda**b**cd**
$\{a^n b^{3n} \mid n > 0\}$	aab**bbbbba**aab**bbbbbbbba**b**bb**
$\{a^n b^m c^{n+m} \mid n, m > 0\}$	aabc**cca**aabbc**cccca**bc**c**
$X \to cXc,\ X \to bXb,\ X \to a$	bca**cb**cbcca**ccbc**bbbcba**bcbbb**
$\{a^n b^m c^{nm} \mid n, m > 0\}$	aabc**ca**aabbc**ccccca**bc

Table 1. Examples generated from the algorithms studied in this paper. In bold, the characters which can be predicted deterministically. During training, we do not have access to this information and at test time, we evaluate only on deterministically predictable characters.

could potentially lead to approaches able to learn small algorithms from sequential data. This can be useful for numerous applications in fields requiring some form of planning. We are aware that the model we propose in this paper is too simple to learn all possible algorithms, but it is interesting to see what problems can possibly be solved.

Our definition of a stack in a recurrent net is through constraining part of the recurrent matrix, similar to (Mikolov et al., 2014) where it was shown that diagonal recurrent matrix can help the recurrent net to store longer memory. In case of a stack, we show that simple structural constrains can allow the network to operate as if it did perform the `PUSH` and `POP` stack operations.

Our work can be seen as a follow up of the research done in early nineties, when similar types of stack RNNs were studied (Pollack, 1991; Das et al., 1992; Mozer & Das, 1993; Zeng et al., 1994). Among recent papers with similar motivation, we are aware of the Neural Turing Machine (Graves et al., 2014) and Memory Networks (Weston et al., 2014). In our work, we tried to isolate the fundamental problems, and solve them in a general way, using standard optimization tools such as stochastic gradient descent and discrete search algorithms.

2. Algorithmic Patterns

We are interested in sequences generated by simple short algorithms and our goal is to learn these algorithms by performing sequence prediction. We are mostly interested in discrete patterns that seem to be related to those that occur in the real world, such as various forms of a long term memory.

More precisely, we suppose that we have only access to a long stream of data which is obtained by concatenating sequences generated by a given algorithm. We do not have access to the boundary of any sequence nor to sequences which are not generated by the algorithm. We denote the regularities in these sequences of symbols as *Algorithmic patterns*. In this paper, we focus on algorithmic patterns which involve some form of counting, addition, multiplication and memorization. In Table 1, we show some examples of these patterns as well as the stream of data obtained by concatenating them. In particular, for memorization we use the context free grammar defined by the following rules: $X \to bXb$, $X \to cXc$, $X \to a$. The description of an algorithm can often be given in a form of a context free grammar, however in general we are interested in sequential patterns that have a short description length in some general Turing-complete computational system. However, we restrict our study to patterns which require only one computational step to predict each symbol. For example, we do not consider algorithms such as sorting of continuous numbers.

We use the unary numeral system to represent algorithms such as counting, addition or multiplication. This allows us to focus on designing a model which could learn these algorithms if the input is given in its simpler form. Learning encoder and decoder from other numeral system to the unary one is out of the scope of this paper.

3. Related work

The algorithmic patterns we study in this paper are closely related to context free and context sensitive grammars which were widely studied in the past. Some works used recurrent networks with hardwired symbolic structures (Grünwald, 1996; Crocker, 1996; Fanty, 1994). These networks are continuous implementation of symbolic system, and can deal with recursive symbolic systems in computational linguistics. While theses approaches are interesting to understand the link between symbolic systems and neural networks, they are often hand designed for each specific grammar.

Wiles & Elman (1995) has shown that it was possible to learn a standard recurrent network able to count on a limited range. They train a recurrent network on sequences of the form $a^n b^n$ for $n = 1$ to 12 and it is able to generalize up to $n = 18$. While this is a promising result, their network has relatively small capacity which makes it hard to generalize beyond $n = 18$. This suggests that their neural network does not truly learn how to count but instead remember seen patterns. Rodriguez et al. (1999) further studied the behavior of this network. Grünwald (1996) designs a hardwired second order recurrent network to tackle sequences of the form $(ab^k)^n$. They show that they are able to tackle sequences up to $k = 120$. Their work is purely exploratory and does not discuss if it is possible to learn such regularities directly from sequences. Christiansen & Chater (1999) extended these results to grammars with larger vocabularies. When the sequences relies on more than two symbols, the network must learn dependencies between its internal representation of the symbols. While they show that recurrent networks have the capacity to learn more complex sequences, their model does not have the capacity to generalize to longer sequences generated by the same algorithm. They also focus on mirror recursive grammars, which we also study.

Beside using standard recurrent networks, other structures have been used to deal with recursive patterns. For example, Tabor (2000) uses a model based on pushdown dynamical automata and Bodén & Wiles (2000) use a sequential cascaded network (Pollack, 1991). More recently, Graves et al. (2014) uses a model inspired by a Turing machine which obtains promising results on challenging algorithmic patterns. In our work, we are interested in learning similar algorithmic patterns with a simpler model. Zaremba & Sutskever (2014) also show that they can learn simple algorithms which involve addition and memorization. They use Long Short Term Memory network (LSTM, Hochreiter & Schmidhuber (1997)) which shows interesting results on these tasks. They use the same distribution of algorithmic patterns for training and testing, and it is not clear how LSTM would generalize to patterns produced from another distribution. For example, sequence memorization requires allocating memory which can be hard to accomplish with a fixed size model.

Closer to our approach, many works have used external memory modules with a recurrent network, such as stacks (Das et al., 1992; 1993; Zeng et al., 1994; Holldobler et al., 1997; Mozer & Das, 1993). Zeng et al. (1994) uses a discrete external stack which may be hard to learn on long sequences. Das et al. (1992) learns a continuous stack which has some similarity with ours. However their models do not seem to be able to generalize to long sequences. Also, as in (Das et al., 1993; Mozer & Das, 1993), their model is train on supervised data, while we aim at learning on the more challenging task of sequence prediction.

Finally, our stack representation is related to the multiplicative gate mechanism used in LSTM and the Gated Recurrent Unit (GRU, Cho et al. (2014); Chung et al. (2015)). A major difference though is that our model can potentially store infinite memory, and does not have to "erase" its previous memory to store a new unit.

4. Model

4.1. Simple recurrent network

We consider sequential data that comes in the form of discrete tokens, such as characters or words. We suppose that these sequences are generated by an algorithm and concatenated to form a long stream of data. For example, given algorithmic patters of the form $\{a^n b^n \mid n > 0\}$, a possible stream of data would be `aabbaaabbbab`. Our goal is to design a model able to predict the next symbol in these streams of data. Our approach is based on a standard model called recurrent neural network (RNN) and popularized by Elman (1990).

A RNN consists of an input layer, a hidden layer with a recurrent time-delayed connection and an output layer. The recurrent connection allows the propagation through time of information about the state of the hidden layer. Given a sequence of tokens, a RNN takes as input the one-hot encoding x_t of the current token and predicts the probability y_t of next one. Between the current token representation and the prediction, there is a hidden layer with m units which stores additional information about the previous tokens seen in the sequence. More precisely, at each time t, the state of the hidden layer h_t is updated based on its previous state h_{t-1} and the encoding x_t of the current token, according to the following equation:

$$h_t = \sigma\left(Ux_t + Rh_{t-1}\right), \tag{1}$$

where $\sigma(x) = 1/(1 + \exp(x))$ is the sigmoid activation function applied coordinate wise, U is the $d \times m$ token embedding matrix and R is the $m \times m$ matrix of recurrent weights. Given the state of these hidden units, the network then outputs the probability vector y_t of the next token, according to the following equation:

$$y_t = f\left(Vh_t\right), \tag{2}$$

where f is the softmax function and V is the $m \times d$ output matrix, where d is the number of different tokens. This architecture is able to learn relatively long complex patterns similar in nature to the ones captured by N-grams. While this has made them interesting for language modeling (Mikolov, 2012), it may not have the capacity to learn how the algorithmic patterns are generated. For example, an RNN with enough capacity - or hidden units - will be able to learn sequences of the form $a^n b^n$ up to the maximum n seen in the training set, but it will not be able to generalize to longer sequences. Basically instead of learning to count, the RNN will tend to memorize all patterns in a similar way N-grams would do.

A possible solution to this problem is to add linear hidden units to the RNN which can potentially remember the length of the sequence. However, linear hidden units are not simple to learn in a recurrent network because of the exploding gradient problem (Bengio et al., 1994). In the next section, we follow another direction introduced in Das et al. (1992): we use a continuous version of a pushdown stack as an external memory which has the theoretical capacity to learn simple algorithmic patterns.

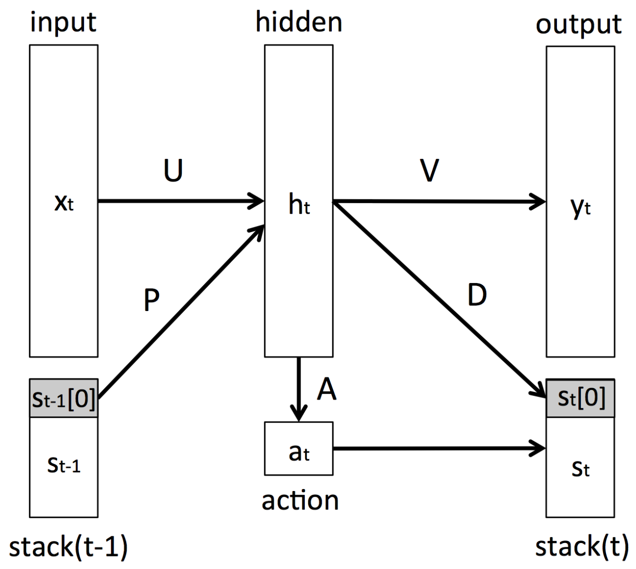

Figure 1. Neural network extended with push-down stack and a controlling mechanism that learns what action (among PUSH, POP and NO-OP) to perform.

4.2. Pushdown network

Our simple algorithmic patterns are very similar to context free grammars (CFGs). These grammars can be recognized by pushdown automaton, i.e., automaton which employs a stack. To design a RNN based model able to learn our algorithmic patterns, it seems natural to design a stack-like structure which can be learned from the data. This type of model has been widely used in the early nineties with some encouraging success (Das et al., 1992; 1993; Zeng et al., 1994; Holldobler et al., 1997; Mozer & Das, 1993). In this section, we propose a novel approach based on similar ideas. We propose two optimization schemes: stochastic gradient descent (Werbos, 1988). and a combination of stochastic gradient descent and search based optimization.

Stack is a type of persistent memory which can be only accessed through its topmost element. Two basic operations can be done with a stack: POP removes the top element and PUSH adds a new symbol on top of the stack. At any given time, the model can thus choose (i) to push a new element on the top of the stack, (ii) to pop the element on the top of the stack, or (iii) do nothing. For simplicity, at first we consider a version where the model has to perform either a PUSH or a POP at any given time. We suppose that this decision is made by a 2-dimensional variable a_t depending on the state of the hidden variable h_t. More precisely, we have:

$$a_t = f\left(Ah_t\right), \qquad (3)$$

where A is a $2 \times m$ matrix and f is a softmax function. We denote by $a_t[\mathtt{PUSH}]$, the value of the PUSH action, and $a_t[\mathtt{POP}]$ the value of the POP action. Note that $a_t[\mathtt{PUSH}] + a_t[\mathtt{POP}] = 1$.

We suppose that our stack is stored at time t in a vector s_t of size p. Note that p could be increased on demand and does not have to be fixed. The top element is stored at position 0, with value $s_t[0]$. Depending on the action variable a_t, the top element will be replaced by either a non-linear transformation of the hidden layer h_t (in case of PUSH) or the value stored below it (in case of POP). More precisely:

$$s_t[0] = a_t[\mathtt{PUSH}]\sigma(Dh_t) + a_t[\mathtt{POP}]s_{t-1}[1], \qquad (4)$$

where D is $1 \times m$ matrix and $\sigma(x)$ is the sigmoid function. If $a_t[\mathtt{POP}]$ is equal to 1, the top element is replaced by the value below (all values are moved by one position up in the stack structure). If $a_t[\mathtt{PUSH}]$ is equal to 1, we move all values down in the stack and add a value on top of the stack. More precisely, for an element stored at a depth $i > 0$ in the stack, we have the update rule:

$$s_t[i] = a_t[\mathtt{PUSH}]s_{t-1}[i-1] + a_t[\mathtt{POP}]s_{t-1}[i+1]. \qquad (5)$$

We use the stack to carry information to the hidden layer at the next time step. When the stack is empty, s_t is set to -1. The hidden layer h_t is now updated as:

$$h_t = \sigma\left(Ux_t + Rh_{t-1} + Ps_{t-1}[0]\right), \qquad (6)$$

where P is a $m \times k$ recurrent matrix, where k is the depth used in the stack to predict the next hidden. In this paper, we focus on understanding how to store memory with a stack. We thus assume for the rest of this paper, that the recurrent matrix R is equal to 0, as shown in Figure 1.

Stack with a no-operation. It is straight-forward to extend our model to deal with an additional action NO-OP. We simply extend the action vector by one element. Then, the stack update rule also allows to not change the context of the stack. For example, Eq. (4) is replaced by:

$$s_t[0] = a_t[\mathtt{PUSH}]\sigma(Dh_t) + a_t[\mathtt{POP}]s_{t-1}[1] + a_t[\mathtt{NO\text{-}OP}]s_{t-1}[0],$$

with $a_t[\mathtt{PUSH}] + a_t[\mathtt{POP}] + a_t[\mathtt{NO\text{-}OP}] = 1$.

Extension to multiple stacks. Using a single stack has serious limitations, especially considering that at each time step, only one action can be performed - the model cannot for example push two values in one simulation step. While the simple model with a single stack is sufficient to learn trivial algorithmic patterns, it is necessary to use more powerful model to deal with more challenging ones. One way to solve this problem would be to learn how many steps need to be performed before the computation is finished. This would allow for example to push two or more values on one stack. In this paper, we use a simpler way to increase capacity of the model, which is to use more stacks in parallel. The stacks interact through the hidden layer allowing them to process more challenging patterns.

Training with SGD. The model presented above is continuous and can thus be trained with stochastic gradient descent (SGD) method and back-propagation through time (Rumelhart et al., 1985; Williams & Zipser, 1995; Werbos, 1988). We use gradient clipping to avoid gradient explosion (Mikolov, 2012). The details of the implementation are given in the experiment section.

Discrete model. Continuous stacks may introduce small imprecision which could cause numerical issues when we work on very long sequences. Also, using a continuous representation is slower at test time as it requires to update the whole stack during the forward propagation. In the result section, we discuss the benefit of discretizing our model at test time. Even though we learn a continuous model during training, we observe that this discretization seems to help in many cases.

This suggests to use a discrete model also during training. In the next section, we propose a simple discrete optimization scheme to train our model.

4.3. Search-based learning and combination with SGD

Training the above recurrent stack network with SGD seems to be an elegant solution. However, for the type of problems that we are interested in this paper, it seems that SGD may not be adapted: ultimately, we want to learn how to operate stack-based long term memory, and the `PUSH` and `POP` operations are in principle discrete. Thus, there is a significant risk that for non-trivial problems, SGD based training will always get stuck in local minima. An example of such unwanted behavior may be learning a bigram model first (understanding the frequency of symbol co-occurrence), which may not be an optimal solution. Every subsequent attempt to move outside of such solution may be prevented by the nature of SGD.

To avoid getting stuck in local minima where the chance of finding better solution becomes zero, one can consider adding random noise to the training examples. A more principled approach would be to include a search based strategy. As a model trained with SGD can only follow one path during its optimization (it has only one set of weights), any non-linear choice made during optimization of the parameters may prevent finding a good local minimum. Using search with many models, we may expect that different models will choose to explore different parts of the search space, and on average should find much better solutions than pure SGD.

Obviously, the size of the search space increases exponentially even for a modest number of possible actions. For our simple experiments, we studied simplified stack models that uses `MAX` function over push 0 (`PUSH0`), push 1 (`PUSH1`) and `POP` actions. Thus, it can be seen as a discrete version of the model described in the previous section. At every step, it executes exactly one action. Instead of having just one model, we keep a pool of models (we use 100). The mapping from the hidden layer to the output can easily be learned with gradient descent. However, this cannot be done for the action outputs - we do not have the targets, and the choice of actions has very non-linear behavior that often influences the state of the stacks far into the future.

To obtain the target for the action outputs, we use a strategy inspired by reinforcement learning (Sutton & Barto, 1998): we sample it using the probability distribution over actions computed by the current model. This stochastic choice quickly makes the models different, which is exactly what we aim for - diversity seems crucial for any efficient search strategy. We continue sampling the targets for several time steps (30 in our experiments; using significantly more steps would make the models less diverse, following the law of big numbers). After that, we train all models using just targets for the output predictions for another 2000 steps. Then, we run evaluation of all models on novel data, again using 2000 steps, to see which model performs the best. All models that have below average performance are replaced in the next training epoch by the best performing model. The last tricks are to keep one copy of the best model unchanged, and train one copy with just SGD and no target action sampling.

Such training has one important advantage over SGD: it is stable. In principle, the performance cannot degrade (if we would evaluate the models on infinitely large validation set to pick the best performing one). Also, guiding the search based on the learned distribution over actions did prove important in our experiments which will be described in the next section.

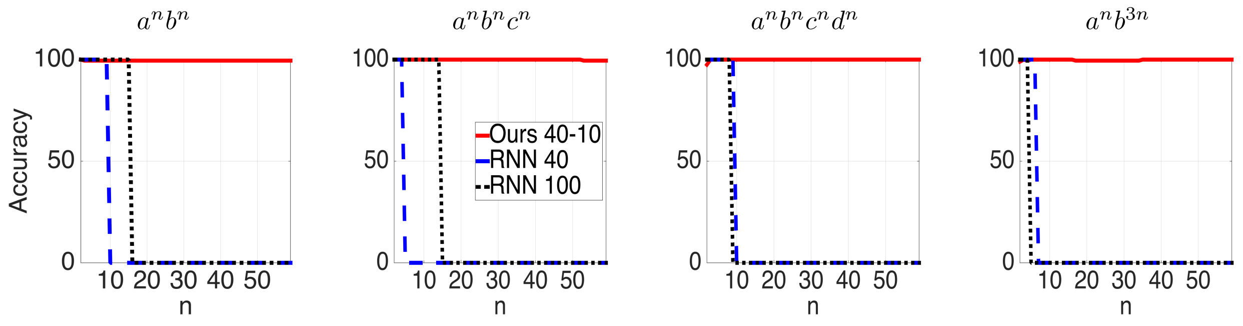

Figure 2. Comparison between our model and RNN on sequences generated from "counting" algorithms. The accuracy is in the proportion of correctly predicted sequences generated with a fixed n.

5. Experiments and results

We evaluate our model on two problems. In the first one, we consider different simple sequences generated by simple algorithms and the goal is to learn their generation rule. We consider similar patterns to the one studied previously (Das et al., 1992; Rodriguez et al., 1999; Bodén & Wiles, 2000).

In the second experiment, we consider the synthetic tasks introduced in Weston et al. (2014). In these tasks, we have a finite world of 4 characters, 3 objects and 5 rooms. Each task is a story about this finite world and the goal is to answer questions of increasing complexity about the state of the world. These tasks have been designed to test the capacity of a model to hold discrete information about this synthetic world for long period of time.

Implementation details. We implemented the model following the graph in Figure 1. The recurrent matrix R defined in Eq. (1) is set to 0. When trained with SGD and backpropagation through time with 50 steps (Werbos, 1988), we use an hard clipping of 15 (Mikolov, 2012), and an initial learning rate of 0.1. We decay our learning rate by multiplying by 0.999 each time the entropy on the validation set is not decreasing. We use a depth k (defined Eq. (6)) of 2 for our stacks during the experiments. The only free parameters are the number of hidden units, stacks and if we use the NO-OP operation. Our implementation is in C++ and runs on on a single core machine.

5.1. Learning simple algorithmic patterns

We consider a set of patterns generated by algorithms with short description length. We generate sequences from a given algorithm and concatenate them into longer sequences. This is thus an unsupervised task, i.e., the boundaries of each generated sequences are not known. We are studying patterns which are related to simple algorithms such as counting, memorization, addition and multiplication. We show examples of generated sequences in Table 1. Our goal is to evaluate if a model has the capacity to understand the generation rule used to produce the sequences. We thus use at test time sequences of size it has never seen before during training. More precisely our experimental setting is the following: We train our model on sequences generated with n up to $N < 20$. We use a validation set made of sequence generated with n up to 20 and we test on sequences generated with n up to 60. We vary N for the different algorithmic patterns and approaches and keep the one with the lowest entropy on the validation set. During training, we use 1000 randomly generated sequences at each epoch, and we incrementally increase the parameter n every few epochs until it reaches 20. We find that this incremental strategy allows both our model and RNNs to learn first simple patterns and then usually generalize them better (Elman, 1993). At test time, we measure the performance by counting the number of correctly predicted sequences. A sequence is considered as correctly predicted if we correctly predict its deterministic part, shown in bold in Table 1.

We compare our model to two RNN models, one with 40 hidden units (RNN 40) and one with 100 (RNN 100). For most of the experiments, our model has 40 hidden units and 10 stacks (Ours 40-10), with a depth of $k = 2$ and with only PUSH and POP actions.

Counting. We show results on patterns generated by "counting" algorithms in Figure 2. The performances are evaluated for fixed n from 2 to 60. As we can see that both RNNs overfit on the training set, and are not able to generalize to longer sequences. Note that if we would use linear hidden units in RNN, it would be in theory possible to make them work on these sequences, but learning a linear RNN is quite challenging because of the exploding gradient problem (Bengio et al., 1994). On the other hand, our model is able to generalize on all of these problems. Despite having a comparable number of parameters with RNN 40, our model is able to avoid overfitting. We show in Table 3 an example of actions done by our model with a single stack (Ours 10-1) on sequences of the form $a^n b^n$. For clarity, we show an example on sequences generated with n equal to 3 and 4, and we use discretization at test time. We can see that, our model learns to push an element when it sees an a and to pop one when it sees a b. At the end of the sequence, the stack is almost empty and with an access to a depth of $k = 2$, it is able to predict the end of the sequence. In Table 4, we also show an example of actions done by our model on sequences of the form $a^n b^{3n}$. These sequences cannot be learn with a single stack and require interactions between at least two stacks to be solved. For clarity we use only two stacks (and 20 hidden units) and discretization on a sequence generated with $n = 3$. We see that the two stacks are able to interact to correctly predict the deterministic part of the sequence, as well as when the sequence ends.

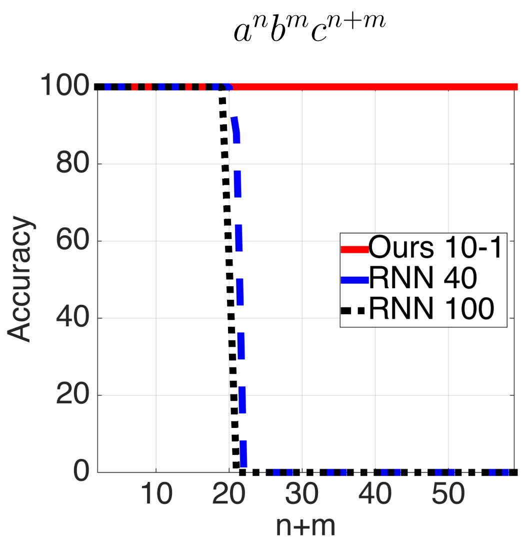

Figure 3. Comparison between our model and RNN on the addition, i.e., $\{a^n b^m c^{n+m} \mid n, m > 1\}$. The accuracy is in the proportion of correctly predicted sequences generated with a fixed $n + m$.

Other algorithmic patterns. We also consider patterns related to addition, multiplication and memorization. We consider sequences of the form $a^n b^m c^{n+m}$ of addition and $a^n b^m c^{nm}$ for multiplication. For memorization, we consider sequences generated from a mirror recursive grammar (Christiansen & Chater, 1999). This sequences are generated by the rules: $X \to bXb$, $X \to cXc$ and $X \to a$. The character a plays the role of a delimiter. We show an

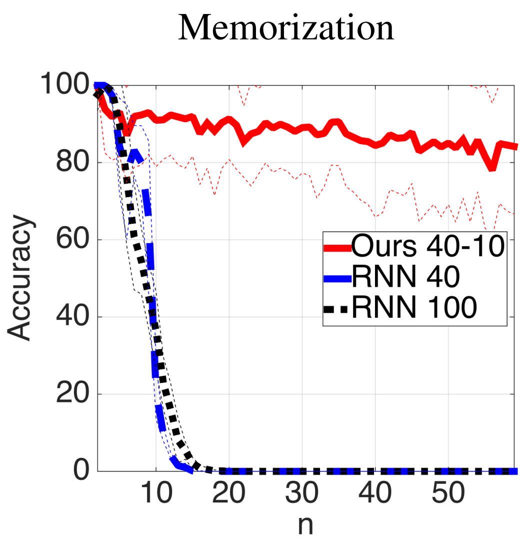

Figure 4. Comparison between our model and RNN on the mirror recursive grammar (Christiansen & Chater, 1999). The accuracy is in the proportion of correctly predicted sequences generated with a fixed n.

example of sequences generated by this grammar in Table 1. Figure 3, we show that our model works well for addition, which is not too surprising since this is close to counting. Note however, that we use only one stack and 10 hidden units, because our model with 40 units with 10 stacks is overfitting as shown below. For memorization and multiplication, we observe some variances in our performances, we thus train 10 different models, and keep 5 with the lowest entropy on the validation set. We show their average performances on the test set in Figure 4 for memorization and Figure 5 for multiplication. On memorization, we see that our model is able to learn this task even though it does not generalize as well as for counting or addition. Note that for memorization, we use smaller epochs of 100 sequences and for multiplication, we use the `NO-OP`.

On multiplication, we see some limitation of our model. It seems that it is not able to generalize much on this task. Our model is marginally better than RNNs but is still overfitting on the training set. For many difficult tasks, we observe that both RNN and our model are quite sensitive to the initialization. For example, it seems that for both memorization and multiplication, we find that three out of ten trained models learn with SGD converged to the same solution than the one that a simple feedforward network (or bigram model) would find. This suggests that for this type of discrete noiseless tasks, learning our model with only SGD may be a problem.

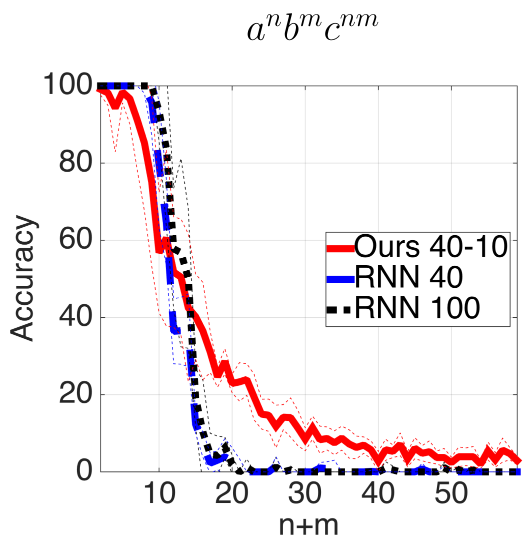

Figure 5. Comparison between our model and RNN on an multiplication, i.e. $\{a^n b^m c^{nm} \mid n, m > 1\}$. The accuracy is in the proportion of correctly predicted sequences generated with a fixed $n + m$.

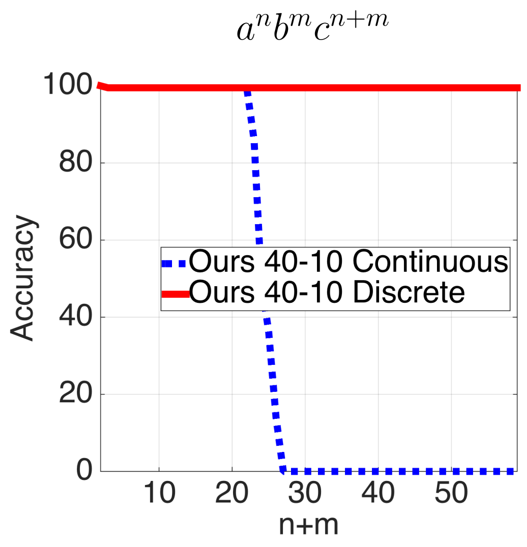

Figure 6. Comparison between our continuous and discrete model on addition, i.e. $\{a^n b^m c^{n+m} \mid n, m > 1\}$. The accuracy is in the proportion of correctly predicted sequences generated with a fixed $n + m$. We discretize our model for validation and test.

Discretization. Using continuous actions and stacks seems to overfit sometimes on the training set. A simple solution is to discretize them during validation to pick models which are using their memory in a discrete way. Figure 6, we show that this simple trick can help for some sequences. This result suggest that using discrete models may lead to more stable solution.

Alice went to the office, and then
Bob moved to the garden, later
Carol went to the kitchen, then
Alice moved to the bedroom.
where was Alice before the bedroom? **Office**.

Figure 7. Example of question answering from Weston et al. (2014). In bold, the answer.

Comparison between SGD and search + SGD. A simple way to break our model is to consider sequences where there is an unbalanced number of as and bs in the sequence, e.g., $a^n b^{6n}$. This is a standard problem in supervised classification with unbalanced number of training example per class. When the statistics of the training data are known, a natural solution is to reweigh each class (e.g., tf-idf for text). However in our case, we aim at learning sequences online with potential change in their statistics, we thus cannot use such solution. On the other hand, the search algorithm described above seems to avoid this problem by exploring a larger set of possible combination of discrete actions. While our study is still preliminary, using a search algorithm on top of SGD allows us to solve sequences such as $a^n b^{4n}$ or $a^n b^{6n}$, where training our model with SGD fails.

5.2. Simulated world question answering

In this section, we consider the question answering tasks proposed in Weston et al. (2014). They propose a simulated world containing 4 characters, 3 objects and 5 rooms. At each time step, a character perform an action - moving around, picking up an object or dropping it. The actions are written into a text using simple automated grammar and every so often, a question is asked about the location of a person or an object. There are two types of questions: either the question concerns the current location of an entity or it concerns a location previously visited, as shown in Figure 7.

These tasks require the models to be able to store information for potentially long period of time. It also requires to store information about multiple objects and persons. Typically, this means that for our model, it will requires a large amount of stacks. These tasks require that we keep information for a long period of time, we thus use the `NO-OP` operation.

In Table 2, we compare our model to RNN, the Longer short term memory model (LSTM, Hochreiter & Schmidhuber (1997)) and the memory network (memNN, Weston et al. (2014)). memNN is a supervised method which used annotation about the support sentences to answer the ques-

	people	people, object	people before	people object before
Unsupervised methods				
LSTM (from Weston et al. (2014))	100%	70.0%	64.8%	49.1%
RNN 100	99.8%	68.3%	59.4%	37.3%
Ours 100-40	100.0%	74.7%	65.8%	51.6%
Supervised methods				
Weston et al. (2014) k=1	97.8%	30.5%	31.0%	24.0%
Weston et al. (2014) k=2, time	100.0%	100.0%	100.0%	100.0%

Table 2. Comparison with RNN and memNN on question answering tasks of Weston et al. (2014).

current	next	prediction	proba(next)	action	stack[top]	stack[top-1]	stack[top-2]
b	a	a	0.99	POP	1	-1	-1
a	a	a	0.90	PUSH	-1	-1	-1
a	a	a	0.94	PUSH	1	-1	-1
a	a	a	0.65	PUSH	1	1	-1
a	b	a	0.34	PUSH	1	1	1
b	b	b	1.00	POP	1	1	1
b	b	b	1.00	POP	1	1	1
b	b	b	1.00	POP	1	1	-1
b	a	a	0.94	POP	1	-1	-1
a	a	a	0.90	PUSH	-1	-1	-1
a	a	a	0.94	PUSH	1	-1	-1
a	b	a	0.34	PUSH	1	1	-1
b	b	b	1.00	POP	1	1	1
b	b	b	1.00	POP	1	1	-1
b	a	a	0.94	POP	1	-1	-1

Table 3. Example of our model with one stack on 2 sequences of the form $a^n b^n$. The stacks stores value between 0 and 1, and the value -1 means that it is empty. We show the 3 first elements of the stack. The stack pushes a value when the input is a a, and pops it when the input is a b, allowing it to store the size of the sequence.

tions. On the other hand, LSTM, RNN and our model are unsupervised. For LSTM, we use the numbers published in Weston et al. (2014). We see that our results are similar to the one obtained by LSTM or RNN. This suggest that our model is basically using its stacks in a similar way as the recurrent matrix of an RNN. Note that our model has less parameters than RNN or LSTM, since we set the recurrent matrix R to 0. Like LSTM, our model fails to capture long discrete patterns even in a relatively low noise setting. We think that there are two reasons for this result: first the number of combinations of entities and places in this database pushes the limit of our model representation power. Second, we think that our model struggles to store complex information required to solve these tasks. This suggests that more complex structures than stacks may be required for this type of tasks.

6. Discussion and future work

In this section, we discuss the achieved results, limitations of our model and potential directions to solve them.

Continuous versus Discrete model and search. In our paper, we show that certain simple algorithmic patterns can be efficiently learned using continuous optimization technique (stochastic gradient descent) and a continuous model representation (in our case a RNN). Note that our model works much better than prior work based on RNN from the nineties that attempts to solve problems of similar type (Das et al., 1992; Zeng et al., 1994; Wiles & Elman, 1995). At the same time, our model seems simpler than many other models used for this type of tasks (Tabor, 2000; Bodén & Wiles, 2000; Graves et al., 2014). Notably, our model is relatively robust and works for a wide range of hyper-parameters (number of stacks, size of the hidden layer or learning rate).

At the same time, we believe that using a continuous representation and an non-convex continuous optimization approach are not the right tools to properly learn algorithms. It seems more natural to attempt to solve these problems with a discrete approach, such as for example a search based approach. This motivated our approach where we combined the continuous and discrete optimization, which

current	next	prediction	proba(next)	action		stack1[top]	stack1[top-1]	stack2[top]	stack2[top-1]
b	a	a	0.99	`POP`	`PUSH`	-1	-1	0	0
a	a	a	0.99	`PUSH`	`PUSH`	-1	-1	0	0
a	a	a	0.93	`PUSH`	`PUSH`	1	-1	0	0
a	b	b	0.12	`PUSH`	`PUSH`	0	1	1	0
b	b	b	1.00	`POP`	`PUSH`	0	0	1	1
b	b	b	0.99	`POP`	`PUSH`	0	1	1	1
b	b	b	0.99	`POP`	`PUSH`	1	-1	1	1
b	b	b	0.99	`POP`	`POP`	-1	-1	1	1
b	b	b	0.99	`POP`	`POP`	-1	-1	1	1
b	b	b	0.99	`POP`	`POP`	-1	-1	1	1
b	b	b	0.99	`POP`	`POP`	-1	-1	1	1
b	b	b	0.99	`POP`	`POP`	-1	-1	1	0
b	a	a	0.99	`POP`	`PUSH`	-1	-1	0	0

Table 4. Example of our model with two stacks (Ours 20-2) on a sequence of the form $a^n b^{3n}$, with $n = 3$. The stacks pushes 0 or 1. The value -1 means that the stack is empty. We show the 2 topmost elements of the stacks. We see that the first stack pushes an element every time it sees a a and pop when it sees b. The second stacks pushes when it sees a a. When it sees a b , it pushes if the first stack is not empty and pop otherwise. This shows that the two stacks has found a way to interact.

allowed us to solve certain problems that seemed to be too difficult for purely continuous optimization.

It is possible that the future of learning of algorithmic patterns will involve such combination of discrete and continuous optimization. However, more work is needed to understand better the limitations of both approaches, and to understand better how to combine them effectively.

Stack-based long-term memory. While in theory using multiple stacks for representing memory is as powerful as a Turing complete system (as we can simulate the tape with just two stacks), complicated interactions between stacks need to be learned to capture more complex algorithmic patterns. For example, certain operations such as accessing distant elements in the memory can be performed much more simply with RAM (random access memory) than with a stack-based memory. It would be interesting to consider in the future other forms of memory which may be more flexible. For example, Weston et al. (2014) and Graves et al. (2014) use more advanced forms of a long term memory.

Finally, complex algorithmic patterns can be more easily learn by composing simplier algorithms. Designing a model which possesses a mechanism to compose algorithms automatically and training it on incrementally harder algorithms is a very important direction for the future.

Learning with other numeral systems. In this paper, we focus on the unary numeral system and we show that given this simple numeral representation, our model can learn simple algorithms. In theory it would be possible to extend our model to other numeral systems by adding an encoder before the input of our network and a decoder after the output. These encoder and decoder could be learn from the data to transform any numeral system into the unary one. Indeed, in this paper we used a one-hot representation for the discrete symbols for both the input and the output. This type of vectors can be easily replaced by the activation of the last (or first) layer of an encoder (or decoder).

Regularization of the stacks. Following the minimum description length principle, algorithms with a short description length should generalize better to new sequences. One way to encourage our model to learn compact representation would be to add a regularization. For example, this regularization could enforce the model to use a minimum number of stacks.

RNN 40	RNN 50	Ours $40 - 10$
161	155	153

Table 5. Comparison with RNNs on Penn Treebank Corpus. We use the recurrent matrix R in our model. Our model has a depth $k = 1$ to have the same number of parameters as the RNN 50. We use a hierarchical softmax with 100 classes for both RNNs and our model.

Language modeling. We also show that our model can be still applied to standard problems such as statistical language modeling. We compare our model with 40 hidden units and 10 stacks to a RNN with either 40 or 50 hidden units on the Penn Treebank Corpus[1] to see if our model is

[1]The pre-processed dataset is available as part of archive at http://www.fit.vutbr.cz/ imikolov/rnnlm/simple-examples.tgz

robust enough to work on language modeling. We add back the recurrent matrix R to our model (defined in Eq. (1)) and we use the NO-OP action. We use a depth $k = 1$ to have the same number of parameters than the RNN 50. We show the results in Table 5. Our model performs a bit better than an RNN with 40 hidden units but has similar performance with the RNN with 50 hidden units.

Not surprisingly, the model does not learn to use stacks in any interesting way, and rather uses them in a similar way as the hidden layer of RNN is normally used. This is because complexity of the real language is so high that one cannot expect to learn algorithms from such data. We believe that in the future, it will be necessary to start learning the natural language from simple examples, and add the complexity only after the system will understand the basic patterns. As we did show in Section 5.2, even learning of a simple associative memory can be quite challenging task, and our model can be difficult to be trained even on such a low complexity language.

7. Acknowledgment

We would like to thank Arthur Szlam, Keith Adams, Jason Weston, Yann LeCun and the rest of the Facebook AI Research team for their useful comments.

8. Conclusion

We have shown that simple stack-based recurrent net can solve certain basic problems such as a^nb^n and memorization, without any hints (such as where the sequence starts), while using just SGD. The solution generalizes to much larger n than what the model is trained on. This is quite positive result, considering the prior work on this difficult topic.

Nonetheless, the SGD seems to be severely limited and when moving to more complex tasks, we had to combine it with search-based learning to solve more complex tasks of the form a^nb^{kn} which require to learn much more complex manipulation with the stack memory. While we were successful at solving some of these toy problems, our model fails on simple algorithmic patterns such as multiplication and it is clear that fully scalable solution to learning algorithmic patterns in sequential data is still an open problem.

While scaling models to larger sizes and training them on more data is very important, we also believe that questioning the limitation of popular architectures and training algorithms is crucial to advance the field toward artificial intelligence.

References

Bengio, Yoshua and LeCun, Yann. Scaling learning algorithms towards ai. *Large-scale kernel machines*, 34(5), 2007.

Bengio, Yoshua, Simard, Patrice, and Frasconi, Paolo. Learning long-term dependencies with gradient descent is difficult. *Neural Networks, IEEE Transactions on*, 5 (2):157–166, 1994.

Bishop, Christopher M. *Pattern recognition and machine learning*. springer New York, 2006.

Bodén, Mikael and Wiles, Janet. Context-free and context-sensitive dynamics in recurrent neural networks. *Connection Science*, 12(3-4):197–210, 2000.

Bottou, Leon. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pp. 177–186. Springer, 2010.

Breiman, Leo. Random forests. *Machine learning*, 45(1): 5–32, 2001.

Cho, Kyunghyun, van Merrienboer, Bart, Gulcehre, Caglar, Bougares, Fethi, Schwenk, Holger, and Bengio, Yoshua. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

Christiansen, Morten H and Chater, Nick. Toward a connectionist model of recursion in human linguistic performance. *Cognitive Science*, 23(2):157–205, 1999.

Chung, Junyoung, Gulcehre, Caglar, Cho, Kyunghyun, and Bengio, Yoshua. Gated feedback recurrent neural networks. *arXiv preprint arXiv:1502.02367*, 2015.

Ciresan, Dan C, Meier, Ueli, Masci, Jonathan, Gambardella, Luca M, and Schmidhuber, Jürgen. High-performance neural networks for visual object classification. *arXiv preprint arXiv:1102.0183*, 2011.

Crocker, Matthew W. *Mechanisms for sentence processing*. Centre for Cognitive Science, University of Edinburgh, 1996.

Dahl, George E, Yu, Dong, Deng, Li, and Acero, Alex. Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition. *Audio, Speech, and Language Processing, IEEE Transactions on*, 20(1):30–42, 2012.

Das, Sreerupa, Giles, C Lee, and Sun, Guo-Zheng. Learning context-free grammars: Capabilities and limitations of a recurrent neural network with an external stack memory. In *Proceedings of The Fourteenth Annual Conference of Cognitive Science Society. Indiana University*, 1992.

Das, Sreerupa, Giles, C Lee, and Sun, Guo-Zheng. Using prior knowledge in a nnpda to learn context-free languages. *Advances in neural information processing systems*, pp. 65–65, 1993.

Elman, Jeffrey L. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.

Elman, Jeffrey L. Learning and development in neural networks: The importance of starting small. *Cognition*, 48 (1):71–99, 1993.

Fanty, Mark. Context-free parsing in connectionist networks. *Parallel natural language processing*, pp. 211–237, 1994.

Graves, Alex, Wayne, Greg, and Danihelka, Ivo. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

Grünwald, Peter. A recurrent network that performs a context-sensitive prediction task. In *Proceedings of the Eighteenth Annual Conference of the Cognitive Science Society: July 12-15, 1996, University of California, San Diego*, volume 18, pp. 335. Psychology Press, 1996.

Hochreiter, Sepp and Schmidhuber, Jürgen. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

Holldobler, Steffen, Kalinke, Yvonne, and Lehmann, Helko. Designing a counter: Another case study of dynamics and activation landscapes in recurrent networks. In *KI-97: Advances in Artificial Intelligence*, pp. 313–324. Springer, 1997.

Krizhevsky, Alex, Sutskever, Ilya, and Hinton, Geoffrey E. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pp. 1097–1105, 2012.

LeCun, Yann, Bottou, Leon, Bengio, Yoshua, and Haffner, Patrick. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Mikolov, Tomas. *Statistical language models based on neural networks*. PhD thesis, Ph. D. thesis, Brno University of Technology, 2012.

Mikolov, Tomas, Joulin, Armand, Chopra, Sumit, Mathieu, Michael, and Ranzato, Marc'Aurelio. Learning longer memory in recurrent neural networks. *arXiv preprint arXiv:1412.7753*, 2014.

Minsky, Marvin and Seymour, Papert. *Perceptrons*. MIT press, 1969.

Mozer, Michael C and Das, Sreerupa. A connectionist symbol manipulator that discovers the structure of context-free languages. *Advances in Neural Information Processing Systems*, pp. 863–863, 1993.

Pollack, Jordan B. The induction of dynamical recognizers. *Machine Learning*, 7(2-3):227–252, 1991.

Recht, Benjamin, Re, Christopher, Wright, Stephen, and Niu, Feng. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pp. 693–701, 2011.

Rodriguez, Paul, Wiles, Janet, and Elman, Jeffrey L. A recurrent neural network that learns to count. *Connection Science*, 11(1):5–40, 1999.

Rumelhart, David E, Hinton, Geoffrey E, and Williams, Ronald J. Learning internal representations by error propagation. Technical report, DTIC Document, 1985.

Sutton, Richard S and Barto, Andrew G. *Introduction to reinforcement learning*. MIT Press, 1998.

Tabor, Whitney. Fractal encoding of context-free grammars in connectionist networks. *Expert Systems*, 17(1):41–56, 2000.

Werbos, Paul J. Generalization of backpropagation with application to a recurrent gas market model. *Neural Networks*, 1(4):339–356, 1988.

Weston, Jason, Chopra, Sumit, and Bordes, Antoine. Memory networks. *arXiv preprint arXiv:1410.3916*, 2014.

Wiles, Janet and Elman, Jeff. Learning to count without a counter: A case study of dynamics and activation landscapes in recurrent networks. In *Proceedings of the Seventeenth Annual Conference of the Cognitive Science Society*, volume 482, pp. 487. Citeseer, 1995.

Williams, Ronald J and Zipser, David. Gradient-based learning algorithms for recurrent networks and their computational complexity. *Back-propagation: Theory, architectures and applications*, pp. 433–486, 1995.

Zaremba, Wojciech and Sutskever, Ilya. Learning to execute. *arXiv preprint arXiv:1410.4615*, 2014.

Zeng, Zheng, Goodman, Rodney M, and Smyth, Padhraic. Discrete recurrent neural networks for grammatical inference. *Neural Networks, IEEE Transactions on*, 5(2): 320–330, 1994.